MOD1

Illustrare le fasi del processo di sviluppo software

Lo sviluppo del software si divide in più fasi:

1. Studio di fattibilità: consiste in una valutazione preliminare dei costi e dei benefici, che porta ad una definizione preliminare del problema e ad una descrizione di costi e tempi necessari.
2. Analisi dei requisiti: prevede una formalizzazione dei requisiti e una successiva analisi che porta alla redazione di un documento di specifica (da validare da parte del committente). Il documento deve specificare quali siano le funzionalità senza dire come verranno realizzate e rappresenta un contratto tra sviluppatori e committente.
3. Analisi del problema: dalla lettura del documento dei requisiti si descrive l'architettura logica del sistema basata sul modello del dominio. L'architettura logica del sistema deve esprimere fatti il più possibile oggettivi sul problema (non sulla sua soluzione) focalizzando l'attenzione sui sottosistemi, sui ruoli e sulle responsabilità. In sostanza è un modello che descrive la struttura del sistema, il comportamento atteso e le interazioni tra le varie parti.
4. Progettazione: l'architettura del sistema è scomposta in più programmi eseguibili (ognuno descritto dalle sue funzionalità e dall'interazione che ha con gli altri programmi), a loro volta scomposti in moduli (ognuno caratterizzato dalle proprie funzioni e relazioni con gli altri moduli).
5. Realizzazione e collaudo dei moduli
6. Integrazione e collaudo del sistema
7. Installazione e training
8. Utilizzo e manutenzione: la manutenzione può puntare a correggere degli errori non rilevati in fase di collaudo, può mirare ad aggiungere nuovi servizi a quelli già esistenti, oppure a migliorare le caratteristiche già esistenti

Principi programmazione OO - Programmazione basata sugli oggetti

Si introducono gli ADT (dati + codice). Ogni ADT ha un'interfaccia visibile all'esterno tramite cui è possibile accedere ai dati (o ad eventuali funzionalità) che invece sono nascosti. A livello teorico si parla di *information hiding* che viene implementato tramite l'incapsulamento ossia la separazione della parte pubblica e della parte relativa all'implementazione nascondendo le scelte progettuali (spesso mutevoli), minimizzando le modifiche necessarie in fase di manutenzione e migliorando la riusabilità del codice.

La programmazione OO si basa sul concetto di interfaccia e di classe.
Un oggetto è un'entità del programma che possiede:

1. uno stato (inteso come insieme di valori associati ai suoi attributi)
2. un insieme di operazioni che operano sullo stato e forniscono servizi ad altri oggetti
3. un comportamento
4. una identità univoca

Gli oggetti sono raggruppabili in classi, ossia entità che descrivono oggetti con caratteristiche (attributi e operazioni) comuni. A compile time una classe definisce l'implementazione di un ADT, mentre a runtime ogni oggetto è istanza di una classe.

Ereditarietà

Le classi possono essere organizzate in una gerarchia di ereditarietà. È importante che gli oggetti della sottoclasse esibiscano tutti i comportamenti e le proprietà della superclasse (Principio di sostituibilità di

Liskov), ma una sottoclasse può anche esibire comportamenti aggiuntivi o eseguire in maniera differente alcune funzionalità della superclasse. L'ereditarietà migliora l'estendibilità del software in quanto evita di dover replicare codice e permette di ridefinire solamente le caratteristiche specifiche all'interno di una gerarchia.

Esistono più tipi di ereditarietà:

- **ereditarietà di modello**: riflette una relazione "Is a". Rappresenta un meccanismo di compatibilità tra tipi lungo una gerarchia: ereditarietà di interfaccia e ereditarietà di estensione
- **ereditarietà software**: rappresenta relazioni all'interno del software stesso anziché all'interno del modello
- **ereditarietà di realizzazione**: rappresenta un meccanismo di riuso all'interno del software. Usata spesso quando una classe ha bisogno di utilizzare i servizi forniti da un'altra classe. La superclasse definisce parte dell'implementazione della sottoclasse e quindi si rompe l'incapsulamento rendendo più difficile il riuso della sottoclasse.
- **composizione e delega**: anziché far derivare la classe B dalla classe A di cui sono necessari i servizi si crea un attributo privato all'interno di B avente tipo A e si delega a quell'oggetto lo svolgimento del compito. Con questa tecnica, inoltre, il legame avviene a runtime garantendo maggiore flessibilità.

Inoltre si hanno:

- ereditarietà semplice: ogni classe deriva da una e sola superclasse. Si ottiene così una struttura ad albero. Questa tecnica è usata da Java e .NET ad esempio.
- ereditarietà multipla: almeno una classe deriva da più superclassi (esempio C++). La struttura che si ottiene in generale è un reticolo. Questo tipo di ereditarietà porta con sé possibili conflitti di nome tra attributi e metodi ereditati dalle varie classi. Tra le classi di questo tipo di gerarchia possono esistere i vincoli overlapping e disjoint. È sempre possibile riportarsi ad un caso di ereditarietà semplice mediante l'utilizzo di "composizione e delega" o tramite l'uso di interfacce.

Quando viene utilizzata l'ereditarietà dinamica?

L'ereditarietà dinamica ha senso quando abbiamo necessità di rappresentare un oggetto di cui son presenti più tipologie differenti che hanno caratteristiche particolari rispetto alla classe generale. Due metodi per simulare l'ereditarietà dinamica sono:

- Usare una **factory** che, a seconda dei parametri di ingresso forniti, costruisca l'istanza della classe corretta. Questo metodo pone un problema nel caso in cui i parametri della classe sono modificabili e dunque una modifica degli stessi potrebbe causare un cambiamento della classe.
- Usare il **pattern State** per risolvere il problema descritto sopra. Tale pattern, attraverso il meccanismo di composizione e delega, permette di incorporare l'istanza del tipo all'interno della classe generale; nel caso in cui avvenga una modifica ai parametri si controlla se occorre cambiare il tipo della classe e a seconda del tipo saranno accessibili una serie di parametri e metodi propri di quel caso particolare.

Illustrare le relazioni tra classi e oggetti

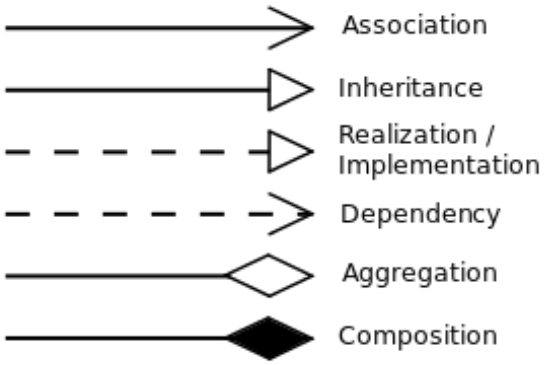
La maggior parte delle classi e degli oggetti interagisce con altre classi ed altri oggetti. Tale interazione è possibile solo se tra le diverse entità sussiste una relazione. Di conseguenza, oltre a modellare le entità coinvolte, è necessario modellare anche le relazioni tra queste entità.

Nella modellazione object-oriented le relazioni principali sono:

- generalizzazione/ereditarietà (*Is a*)
- realizzazione (*implements*): usata dalle interfacce

- associazione
    - generica
    - aggregazione (*has*): relazione non forte in quanto le classi parte hanno significato anche senza che sia presente la classe contenitore, inoltre le classi parte hanno un tempo di vita slegato d quello della classe contenitore (es. un museo contiene quadri e statue).
    - composizione (*has subpart*): relazione forte in quanto le parti hanno senso solo se in presenza della classe contenitore, inoltre le classi parte hanno tempo di vita dipendente da quello della classe contenitore e vengono distrutte alla distruzione del contenitore (es. un animale ha quattro zampe).
- dipendenza
    - collaborazione (*uses*)
    - relazione istanza-classe
    - relazione classe-metaclasse

Queste relazioni possono essere interpretate secondo lo schema cliente/fornitore.



Processo di sviluppo orientato agli oggetti

Si parte da un assunto di base: il mondo è composto di oggetti. Un sistema software rappresenta una porzione del mondo reale (o virtuale) ed è composto anch'esso di oggetti che interagiscono tramite scambio di messaggi.

- Analisi orientata agli oggetti: ha come scopo la modellazione di una porzione del mondo reale. Ogni classe descrive una categoria di oggetti.
- Progettazione orientata agli oggetti: ha come scopo la modellazione della soluzione. In questa fase vengono introdotti gli oggetti di programmazione (algoritmi, strutture dati...). Ogni classe descrive un tipo di dato differente.
- Programmazione orientata agli oggetti: ha lo scopo di realizzare la soluzione tramite l'utilizzo di linguaggi di programmazione OO e di sistemi run-time. Ogni classe descrive l'implementazione di tipo di dato.

Illustrare la struttura a livelli di un'applicazione

Ciò che si trova tra quello che vede l'utente ed i dati fisicamente scritti in memoria può essere suddiviso in vari livelli:

- Presentation Manager: si occupa dell'interazione con l'utente tramite un'interfaccia (GUI o a caratteri)

- Presentation Middleware: software che permette di trasferire i dati da una interfaccia all'applicazione e viceversa (es. software di emulazione di terminali numerici, web browser + web server + HTTP, ...)
- Presentation Logic: gestisce l'interazione con l'utente a livello logico (accettazione dei dati, parziale validazione dei dati, gestione errori...)
- Application Logic: logica dell'applicazione (talvolta separata in più livelli comunicanti per mezzo di un Application Middleware che gestisce la comunicazione tra sue componenti della stessa applicazione)
- Data Logic: gestisce la persistenza a livello logico (consistenza dei dati, gestione degli errori, gestione file/database)
- Database Middleware: software che permette di trasferire i dati dall'applicazione ad un gestore di persistenza e viceversa (es. le richieste SQL dall'applicazione al DBMS, i dati dal DBMS all'applicazione).
- Data Manager: gestisce fisicamente i dati dell'applicazione tramite l'uso di file o di database

I livelli centrali costituiscono l'applicazione vera e propria. Un'applicazione dovrebbe essere il più generale possibile e comunicare in maniera logica con il mondo esterno in modo da essere resistente ai cambiamenti.

Modificatori dei metodi:

- **virtual**: se un metodo è marcato come virtuale la sua implementazione può essere cambiata da una sottoclasse (override) e viene effettuato a run-time un controllo sul tipo dell'oggetto tramite cui è possibile risalire a quale sia l'implementazione specifica di quel metodo (controllo all'interno della VM). Di default in .NET i metodi non sono virtuali: per B -> A, se un oggetto B viene assegnato ad una variabile A, allora viene eseguito il codice di A.
- **abstract**: un metodo astratto non contiene implementazione ed è implicitamente virtuale.
- **override**: i metodi marcati come *override* forniscono una nuova implementazione al metodo ereditato. Perché un metodo sia sovrascrivibile è necessario che la classe base lo dichiari come abstract o virtual.
- **sealed:** il marcatore *sealed* impedisce alle sottoclassi di effettuare l'override di quel particolare metodo.

Costruttore

I costruttori di istanza provvedono all'inizializzazione dei campi di un nuovo oggetto, i costruttori di tipo provvedono all'inizializzazione dei campi statici quando la classe viene caricata in memoria.
In generale il costruttore di tipo è dichiarato *static* (può accedere esclusivamente ai membri statici), implicitamente *private*, senza argomenti e va definito solo se strettamente necessario, ovvero se i campi statici della classe non possono essere inizializzati in linea oppure devono essere inizializzati solo se la classe viene effettivamente utilizzata.
Se si verifica un'eccezione nel costruttore e questa non viene gestita nel costruttore stesso:

- nel caso di costruttori di istanza nessun problema.
- nel caso di costruttori di tipo la classe non è più utilizzabile.

Illustrare il ciclo di vita di un oggetto

Per prima cosa viene allocata la memoria necessaria ad ospitare l'oggetto, successivamente all'interno di quest'area di memoria l'oggetto viene inizializzato. A questo punto l'oggetto è funzionante. In caso l'oggetto non contenga risorse associate al sistema operativo o risorse che vadano rilasciate (file aperti,

socket, connessioni varie...) la memoria occupata dall'oggetto può direttamente essere liberata, rendendola disponibile per altri usi. In caso invece l'oggetto possedesse risorse particolari dovrebbero essere chiuse (l'oggetto dovrebbe essere finalizzato) prima della deallocazione della memoria.

Possibili strategie per il Garbage Collector

- **Tracing**: si determina quali oggetti siano raggiungibili, tutti gli altri possono essere eliminati. Definito un insieme di oggetti detti "radice", si passa ad analizzare iterativamente gli oggetti per determinare se siano raggiungibili. Un oggetto è raggiungibile se:
  - è una radice
  - è raggiungibile ricorsivamente partendo da una radice, ovvero esiste una catena di riferimenti che partendo da una radice conduce all'oggetto in questione.
- **Refernce counting**: ad ogni puntatore viene associato un contatore che indica il numero di riferimenti a esso, la memoria può essere liberata quando il contatore raggiunge lo 0. In caso di riferimenti incrociati diventa inutile, e comunque aumenta l'occupazione di memoria.
- **Escape analysis**: vengono spostati gli oggetti dallo heap allo stack, l'analisi è fatta a compile time per determinare quali oggetti abbiano la necessità di essere accessibili fuori dalla routine che li ha creati. Gli oggetti che si trovano sullo stack vengono deallocati automaticamente all'uscita della routine.

Descrivere il principio zero o rasoio di Occam

Non bisogna introdurre concetti che non siano strettamente necessari / quel che non c'è non si rompe
Tra le varie soluzioni possibili occorre scegliere quella più semplice e quella che si porta dietro meno ipotesi/concetti di base e genera meno dipendenze. La semplicità iniziale va poi mantenuta lungo tutta la durata dello sviluppo.

Passi durante la progettazione di dettaglio

Durante la progettazione di dettaglio è necessario definire:

- Tipi di dato che non sono stati definiti nel modello OOA
- Navigabilità delle associazioni tra classi e relativa implementazione. Riguarda la possibilità di spostarsi efficientemente da un oggetto di origine ad un oggetto di destinazione. Una relazione può essere sia monodirezionale che bidirezionale, in questo ultimo caso va effettuato un controllo sulla consistenza delle strutture dati. Le associazioni con molteplicità 1..1 o 0..1 possono essere risolte semplicemente aggiungendo un attributo membro, per le associazioni con molteplicità 0..* oppure 1..* occorre far ricorso ad una classe contenitore, che possono essere distinte:
  - dal modo con cui contengono gli oggetti:
    - per valore: nel contenitore viene memorizzata una copia del valore e la distruzione del contenitore implica la distruzione dei valori contenuti (gli oggetti originali possono sopravvivere)
    - per riferimento: il contenitore contiene un riferimento all'oggetto, pertanto la distruzione del contenitore non implica direttamente la distruzione degli oggetti contenuti

  dall'omogeneità/eterogeneità degli oggetti contenuti
- Strutture dati necessarie per l'implementazione del sistema
- Operazioni necessarie per l'implementazione del sistema
- Algoritmi che implementano le operazioni
- Visibilità di classi

- Risolvere i casi di ereditarietà multipla tramite composizione e delega oppure tramite l'uso di interfacce
- Miglioramento delle prestazioni

Illustrare VCS (Version Control System)

Questi sistemi si occupano di tener traccia delle varie versioni di uno o più documenti, rendendo possibile la coesistenza di versioni multiple dello stesso documento o di effettuare un rollback ad una versione precedente. Inoltre permette di procedere in parallelo su differenti aspetti della redazione di documenti (a prescindere dal tipo di documento) incrementando la produttività.

Sono presenti alcuni concetti chiave:

- Progetto: insieme di file (di qualsiasi tipo) gestito dal VCS
- Repository: rappresenta il luogo in cui i file, con la loro storia, vengono salvati. Tipicamente è costituito da un server remoto, ma nulla vieta la costruzione di un repository locale
- Cartella di lavoro: contiene una copia locale del contenuto di un repository. Concettualmente è una sandbox in cui lo sviluppatore può precedere con il proprio lavoro. Si definisce *check out* l'operazione che consiste nella copia dei dati dal repository alla cartella di lavoro, mentre si definisce check in l'operazione inversa.
- Revisioni: ogni volta che viene fatto il "check in" di un file ne viene creata una nuova versione, identificata da un valore incrementale. Poiché generalmente le modifiche sono relativamente piccole per efficienza vengono salvate le differenze tra la nuova versione e la vecchia. Se da un lato questo ottimizza l'occupazione dello spazio, dall'altro rende più complesse le operazioni di "check out" e di "check in"
- Ramificazioni (branch): consiste nella creazione di due copie coesistenti di uno o più file. Per evitare sovrascritture dei dati durante il "check in" sono possibili due strategie:
    - Modello Lock-Modify-Unlock: il repository consente ad un solo utente alla volta di modificare la stessa risorsa, che risulta pertanto bloccata per gli altri utenti. Quando l'utente ha finito con le modifiche viene effettuato il "check out" e la risorsa viene sbloccata. Questa tecnica potrebbe causare una linearizzazione del lavoro non necessaria in quanto rende impossibile a due utenti di lavorare contemporaneamente sulla stesso file ma in aree differenti. Inoltre può venir generato un falso senso di sicurezza: ad esempio l'utente A modifica il file f1, e l'utente B modifica il file f2. Singolarmente le due modifiche non presentano conflitti, ma, poiché f1 dipende da f2, il sistema smette di funzionare correttamente.
    - Modello Copy-Modify-Merge: non vengono introdotti dei lock, ogni volta che vengono effettuate delle modifiche ad una risorsa viene effettuato il merge tra le due versioni.

Merging: consiste nell'unire due rami prima separati per ottenere un nuovo unico ramo. L'unione di due rami può avere successo o meno. Perché il merge abbia successo è necessario che non vi siano conflitti all'interno dei file. Un conflitto è rappresentato da una situazione in cui i due rami vanno a modificare la stessa porzione del documento originale, di conseguenza il VCS non è in grado di comprendere quale sia la versione definitiva e chi mantiene il repository deve intervenire a mano. I conflitti rilevati dai VCS sono puramente sintattici e non semantici, perciò è possibile che due modifiche sintatticamente corrette (per il VCS, non per il codice del programma ad esempio) portino ad un conflitto, ad esempio il cambiamento della firma di una funzione non crea conflitto sintattico ma rende impossibile la compilazione del codice che sfrutta la vecchia firma.

I VCS possono essere sia centralizzati sia distribuiti. Nel secondo caso il repository centrale deve sincronizzarsi con i vari client che posseggono una copia locale del repository. Un sistema distribuito inoltre rende più rapide operazioni comuni come commit e ricerca nella cronologia poiché non sussiste la necessita

di comunicare con il server centrale. Quando si vuol far sì che le informazioni contenute nella propria copia locale del repository vengano condivise occorre effettuare un'operazione di "push" (l'operazione duale e detta "pull").
I VCS distribuiti hanno sia pro che contro. Da un lato permettono di lavorare efficientemente offline in una sandbox locale, dall'altro non esiste un concetto forte di "ultima versione" (si può considerare quella contenuta all'interno del repo centrale come l'ultima versione stabile) e si perde la nozione di numeri di revisione in favore dell'uso di "tag" descrittivi.

MVC e MVP

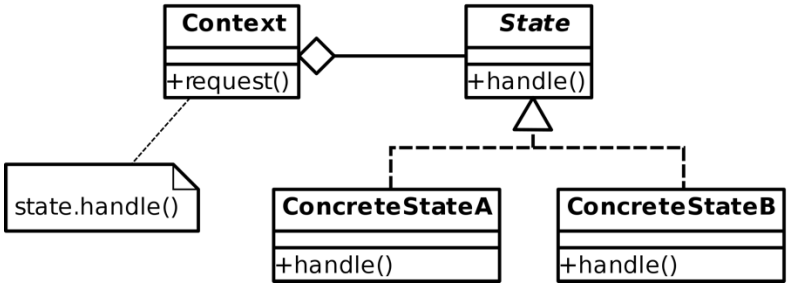
Il pattern MVP `e una versione piu` “incapsulata” del pattern MVC, che favorisce il riuso delle view al costo di maggiore complessit`a.
Modello: gestisce un insieme di dati logicamente correlati; risponde alle interrogazioni sui dati; risponde alle istruzioni di modifica dello stato; genera un evento quando lo stato cambia; registra, in forma anonima, gli oggetti interessati alla notifica dell’evento.
View: gestisce un’area di visualizzazione, nella quale presenta all’utente una vista dei dati fornitele dal presenter; mappa i dati forniti dal presenter, o una parte, in oggetti visuali e visualizza tali oggetti su un particolare dispositivo di output; si registra presso il presenter per ricevere l’evento di cambiamento di stato.
Controller: gestisce gli input dell’utente (mouse, tastiera, etc.); mappa le azioni dell’u- tente nei comandi; invia tali comandi al model, che effettua le operazioni appropriate; aggiorna la view;

Pattern State

Problema: Si vuole che un oggetto cambi comportamento a seconda del suo stato, come se cambiasse classe durante l’esecuzione, cosa non possibile nella maggior parte dei linguaggi a oggetti.

Soluzione: Si associa il contesto a una classe astratta AbstractState, che rappresenta lo stato. AbstractState è implementata dai vari ConcreteState, che hanno un metodo HandleStateChange(). A ogni cambiamento di stato, il contesto chiama Request() che, per implementare il comportamento che dipende dallo stato, a sua volta chiama HandleStateChange() del proprio stato corrente, il quale pu`o a sua volta richiamare un metodo SetState() del cliente per cambiarne lo stato.

Diagramma UML:



MOD2

Definizione di modello

Per modello si intende genericamente una rappresentazione di un oggetto reale che riproduce caratteristiche o comportamenti ritenuti fondamentali o utili per il tipo di ricerca che si sta svolgendo. Per quanto riguarda l'ingegneria del software è un insieme di proprietà e caratteristiche espresse per mezzo di un linguaggio apposito e diagrammi. Lo scopo dei diagrammi è descrivere in maniera concisa e precisa conoscenze sul problema utili in modo da individuare rischi e scelte progettuali. Si usano linguaggi con livello di astrazione più elevato dei linguaggi di programmazione. L'insieme dei modelli, connessi in modo sistematico, deve dare una descrizione completa e non ridondante e la transizione tra un modello e l'altro deve essere continua.
N.B. modello != rappresentazione

Linguaggio di modellazione

È un linguaggio semi-formale che può essere usato per modellare/descrivere un sistema. Tramite tale linguaggio si ottiene una rappresentazione del modello, differente a seconda del linguaggio usato (il potere espressivo dei vari linguaggi può variare), che può essere utilizzata per comunicare tra gli addetti ai lavori in maniera possibilmente precisa e flessibile.
Anche il codice permette di ottenere una rappresentazione di un modello, ma questa rappresentazione ha vari svantaggi:

- è troppo dettagliata
- fornisce una visione piatta
- non mette in evidenza i punti salienti
- non permette di avere rapidamente una visione di insieme

Dato che è necessario avere sia modelli ad alto livello che codice (funzionante) diventa necessario mantenere l'allineamento tra modello e codice (non si usano tecniche di reverse engineering in quanto riproducono nel modello gli stessi problemi del codice stesso). In caso l'allineamento non venga mantenuto viene meno la tracciabilità e i modelli non forniscono più una vista coerente. L'approccio migliore consiste nell'apportare modifiche in primo luogo al modello di progettazione e generare da questo, dove possibile, del codice.

Modello evolutivo di sviluppo software

Si parte dallo sviluppo di un prototipo iniziale che viene a mano a mano raffinato, partendo dalle parti con requisiti più chiari, passando per versioni intermedie in cui si aggiungono i requisiti richiesti dai committenti, fino ad arrivare al prodotto finale. Questo permette di lavorare a stretto contatto con gli utenti in modo da comprenderne al meglio le specifiche e le funzionalità. A mano a mano che le iterazioni si fanno più veloci ci avvicina alla *extreme programming* (XP) cioè si progetta mentre si sviluppa, rende necessario mantenere una certa semplicità nel codice e la scrittura di molti test. A causa delle modifiche frequenti il sistema sviluppato è poco strutturato e non è visibile il processo di sviluppo, perciò questo modello è adatto a progetti di piccole dimensioni o che hanno breve durata.

Come garantire sicurezza informatica

Solitamente l'obiettivo degli attacchi è la ricezione di dati in maniera illecita, perciò obiettivo della sicurezza informatica è la protezione da potenziali rischi e/o violazioni dei dati

- Riservatezza
- Integrità e autenticità
- Disponibilità

Per la protezione è necessario quindi identificare, autenticare ("qualcosa che si è, qualcosa che si ha, qualcosa che si sa") e autorizzare l'utenza che accede ai dati e deve essere garantito un funzionamento affidabile (es. resistenza ad attacchi DoS).

Nella scelta della sicurezza incidono diversi fattori economici e ingegneristici:

- Dinamicità del sistema
- Dimensione del sistema
- Metodologie di accesso
- Tempo di vita delle informazioni
- Costi della protezione
- Costi in caso di violazione
- Valore percepito e tipologia di attaccante

In generale un sistema è forte tanto quanto il suo punto più debole (Catena degli anelli), e la sicurezza va progettata sia a livello fisico che logico (uso di chiavi per cifrare i dati, comunicazioni cifrate, messaggi autenticati (firma digitale), controllo forte degli accessi, controllo forte dei permessi, ...).

Sistemi critici

I Sistemi critici sono sistemi tecnici o socio-tecnici da cui dipendono persone o servizi. Se questi sistemi non forniscono i loro servizi come ci si aspetta possono verificarsi seri problemi e importanti perdite. Ci sono tre tipi principali di sistemi critici:

- sistemi **safety-critical** i cui fallimenti causano perdite umane, incidenti o danni ambientali
- **mission-critical** i cui fallimenti causano problemi all'erogazione dei servizi
- **business-critical** i cui fallimenti causano problemi a delle aziende

Il sistema deve essere disponibile ossia deve poter erogare il suo servizio sempre, affidabile ossia dare gli stessi risultati in caso vengano sottoposti allo stesso stimolo, sicuro ossia garantire una certa sicurezza nell'uso. È necessario tenere conto delle varie cause dei fallimenti hardware, software o comportamento umano.

Nei confronti dei sistemi critici la rete è ambivalente: da un lato permette di distribuire più efficacemente le patch di sicurezza e di divulgare vulnerabilità, ma dall'altro rappresenta una modalità d'accesso al sistema difficile da disciplinare.

Esempi di possibili attacchi sono:

- **exploit**: metodo che sfrutta un bug o una vulnerabilità, per l'acquisizione di privilegi
- **buffer overflow**: fornire al programma più dati di quanto esso si aspetti di ricevere, in modo che una parte di questi vadano scritti in zone di memoria dove sono, o dovrebbero essere, altri dati o lo stack del programma stesso
- **shell code**: sequenza di caratteri che rappresenta un codice binario in grado di lanciare una shell, può essere utilizzato per acquisire un accesso alla linea di comando
- **sniffing**: attività di intercettazione passiva dei dati che transitano in una rete
- **cracking**: modifica di un software per rimuovere la protezione dalla copia, oppure per ottenere accesso ad un'area riservata
- **spoofing**: tecnica con la quale si simula un indirizzo IP privato da una rete pubblica facendo credere agli host che l'IP della macchina server da contattare sia il suo
- **trojan**: programma che contiene funzionalità malevole
- **DoS**: il sistema viene forzatamente messo in uno stato in cui i suoi servizi non sono disponibili, influenzando così la disponibilità del sistema

Nella progettazione di un sistema informatico non si può prescindere dalla componente relativa alla sicurezza, ma il sistema va progettato tenendo conto anche delle minacce a cui andrà in contro. Si parla quindi di Security engineering e comprende ad esempio la gestione degli utenti e di permessi per quanto riguarda la prevenzione di usi illeciti di risorse; l'installazione, la configurazione e l'aggiornamento del software e monitoraggio degli attacchi, rilevazione e ripristino.